Parallel Distributed Algorithms of the β-model of the Small World Graphs[1]

Mahmoud Rafea, Konstantin Popov, Per Brand, Fredrik Holmgren, Seif Haridi

SICS, Box 1263, SE-16429 Kista, Sweden {seif, fredrikh, perbrand, kost, mahmoud}@sics.se

Abstract. The research goal is to develop a large-scale agent-based simulation environment to support implementations of Internet simulation applications. The Small Worlds (SW) graphs are used to model Web sites and social networks of Internet users. Each vertex represents the identity of a simple agent. In order to cope with scalability issues, we have to consider distributed parallel processing. The focus of this paper is to present two parallel-distributed algorithms for the construction of a particular type of SW graph called β-model. The first algorithm serializes the graph construction, while the second constructs the graph in parallel.

Introduction

The idea of small worlds (SW) is based on a phenomenon, which formalizes the anecdotal notion that "you are only ever six 'degrees of separation' away from anybody else on the planet." In this paper, the β-model of SW graphs is going to be described but other graphs are out of the scope. For more information about SW graphs we refer to [6].

The β–model graph is used to model the social network of Internet users where each vertex represents a user and the neighbors of this vertex represent his friends. Each vertex in the graph represents a simple agent [1]. Those agents are used to implement behavior models that simulate emerging phenomena on the Internet. The sequential agent-based simulation environment can be found in [2]. Also, one of the implemented behavior models can be reviewed in [5]. The parallel environment is under publication [3].

The platform is a rack-mounted 16-nodes in a chassis. Each node has: an AMD Athlon (1900+) processor on a Gigabyte 7VTXH+ motherboard, which contains half Gigabyte DDR memory. All the nodes are running Linux and having the same installation. The implementation language is Mozart [4]. The next section (2) introduces the β-model. The motivations of implementing the parallel-distributed graph, approaches and problem are described in section (3). In section (4), the methodology of distributed graphs verification

[1] This work has been done within the *iCities project*, funded by European Commission (Future and Emerging Technologies, IST-1999-11337).

is described. In section (5) the first effort of partitioning the vertexes using a serialized algorithm is described. In section (6), an algorithm for parallel construction of the graph is described.

β -model of SW graph

If u and w are vertexes of a graph G, then an edge of the form (u, w) is said to join or connect u and w. An adjacency list is used to represent the graph. The adjacency list contains all vertexes of the graph and next to each vertex is all the vertexes adjacent to it, is called neighbors. A graph in which all vertexes have precisely the same number of edges (k) is called k-regular or just regular. Examples are demonstrated in **Fig. 1**.

(a)

Id	Neighbors
1	2, 3, 5, 6
2	1, 3, 4, 6
3	1, 2, 4, 5
4	2, 3, 5, 6
5	3, 4, 6, 1
6	1, 2, 4, 5

(b)

Id	Neighbors
1	2, 6
2	1, 3
3	2, 4
4	3, 5
5	4, 6
6	5, 1

Fig 1.: Regular graph representations: (a) N=6, k=4 and (b) N=6, k=2

In β-model, the first step is to construct a regular graph. So any vertex v ($1 \le v \le N$) is joined to its neighbors, u_i and w_i, as specified by: $u_i = (v - 1 - i + N) \pmod N + 1$, AND $w_i = (v - 1 + i) \pmod N + 1$. Where $1 \le i \le k/2$, and it is generally assumed that $k \ge 2$.

The construction of SW graph passes in two phases. The first is the construction of a regular graph. The second is graph rewiring. Rewiring is done for a number of vertexes selected randomly using probability β. It is accomplished by deleting an edge, and inserting a newly randomly generated edge. The new edge can be the deleted edge but never any existing one. A refinement of the algorithm is to store only the rewired vertexes because neighbors can be easily computed using the vertex id and the value of k. Consequently, the algorithm complexity is O(K*N* Beta).

The motivation, approaches and Problems

The motivation of this work is to construct SW graphs with very large number of vertexes, on the scale of 1,000,000. We have used distributed parallel processing to cope with scalability issues. There are two approaches. The first is to serialize the rewiring, which does not need to modify the algorithm of the sequential one. The second is to rewire the graph in parallel, which needs to modify the rewiring algorithm. Implementation wise, the

challenge is to have an efficient messages passing between partitions on the scale of k/2 * β * N * (P-1)/P, where P is number of partitions, and (P-1)/P is the probability to cross the process boundary, e.g., 10/2 * 0.2 * 1000000* 9/10 = 900000 messages.

Algorithm analysis reveals two problems. Consider the regular graph shown in **Fig. 1.** (b). If the vertexes 1, and 4 are going to be rewired concurrently, the edges (1, 2) and (4, 5) will be deleted. The possible edges to insert are: vertex -1 = (1, 2), (1, 3), (1,4), (1, 5); and vertex-4 = (4, 5), (4, 6), (1, 4), (2, 4). The edge (1, 4) can be added by both processes. This creates a graph with a number of edges less than the sequential one (problem 1). Also, using the same regular graph, consider vertexes 3 and 4. The rewiring of 3 may delete the edge (3, 4). This means that while rewiring the vertex 4 in sequence the edge (3, 4) will be considered as one of the possible edges to insert, but this is not the case in parallel rewiring (problem 2).

Distributed graphs verification

The graphs are verified by: (1) checking that the graph is correctly connected, i.e., for all graph vertexes, if vertex w is a neighbor of u, then u must be a member in neighbors of w, (2) counting the total number of edges, which should not change after rewiring, and (3) statistically, comparing graphs that are constructed sequentially with ones that are distributed using the minimum and maximum degree of vertexes; this is needed because rewiring changes a regular graph to an irregular graph. The results reveal that there is no significant difference between them. It should be remarked that random number generator initialization is a key factor to get correct distributed graphs. Initially, we missed this, which leads to graphs with higher maximum value.

Distributed serialized β-model of SW graph

A graph G is partitioned into P partitions with equal size S. The vertexes of the P partitions are: 1 to S, S+1 to 2S, ... and (P-1)*S+1 to P*S. The rewiring is then started in each partition process. Unlike the sequential algorithm, this requires cooperation among processes, which takes the form of remote calls. The parallelization strategy is based on: 1) Detecting the calls that need to cross the process boundary, 2) Collecting those calls in a list, 3) Partitioning the list into sub-lists such that each sub-list contains calls to one process, and 4) Parallel processing of those sub-lists using remote procedure calls (RPC).

This strategy has dual benefits: first, minimizing physical messages sent (RPCs), and second, minimizing thread creation and management. The total number of physical

messages is k/2*P*(P-1). There is only one thread per process that handles the intra-partition rewiring and another thread that handles the inter-partition rewiring.

The rewiring loop is synchronized so that the first loop iteration starts by rewiring the first partition meanwhile; the execution of all other partitions is suspended. Generally, in the same loop iteration, partition (x+1) is suspended; until the rewiring of partition (x) is completed ($1 \leq x < P$). The synchronization data structure is constructed from unbound variables that are organized in tuples. The number of tuples equals the number of partitions (P). The arity of the tuple equals k div 2.

The last part is the edge rewiring code, **Fig. 2**, which is modified so that procedure calls that need to cross the process boundary are handled using the parallelization strategy. In this way, the distributed rewiring behaves exactly similar to the sequential one. Notice that the time complexity is the same as the sequential algorithm.

The experimental study shows that this approach gives the required scalability. On the scale of 2000000 vertexes, there is no plenty on performance. It even performs slightly better. Statistically, this slight improvement is insignificant. Using efficiently hardware resources may explain these results.

```
proc {RewireEdge Id Index}
   OldNeighbor = (Id + Index) mod N
   Neighbors = {Subtract {GetNeighbors Id} OldNeighbor}
   NewNeighbor = {GenerateRandomNeighbor Id Neighbors} in
   if OldNeighbor \= NewNeighbor then
     {RemoveNeighbor Id OldNeighbor}
     {PutNeighbor Id NewNeighbor}
     {RemoveNeighbor OldNeighbor Id}
     {PutNeighbor NewNeighbor Id}
   end
end
```

Fig. 2. : Edge rewiring over distributed partition

Parallel-distributed algorithm

It differs from the serialized algorithm only in synchronization. All RPCs are invoked after the rewiring loop is terminated. This means the number of physical messages, directly generated from rewiring, is only P*(P-1). The edge rewiring is the same, **Fig. 2**, but new procedures for adding and removing edges need to be added to handle the two problems that lead to incorrect graphs, which are described above (section 3).

The first new procedure is called `PutNewNeighbor`/2 (**Fig. 3**). The existence of w in neighbors of u and the condition w < u detects the first problem and generating a new

edge corrects it. Using the same example described in section 4, adding the vertex 1 to the adjacency list of vertex 4 can be detected because the edge already exists and 1 < 4. So, selecting an edge from the possible ones and adding it, is exactly what will happen in the sequential rewiring. Notice that a new edge may cross the process boundary.

```
proc {PutNewNeighbor U W}
 if {AlreadyNeighbor U W} then {PutNeighbor U {GenrateNeighbor U}}
 else {PutNeighbor U W} end
end
```

Fig. 3. : Adding new neighbor in parallel rewiring

The second new procedure is called `RemoveOldNeighbor`/2 (**Fig. 4**). If u has been rewired AND w<u AND u-w≤k/2 detects the second problem. The solution is based on selecting randomly one vertex (v) from newly added neighbors. Removing w and v from the neighbors of u, prepares for a state that is identical to the sequential rewiring. Now the value of a randomly generated vertex, z, will determine the next actions. If z=w, we need to correct the edges, else just remove w from the neighbors of u. To correct the edges: delete edge (u, v) and keep the edge (u, w).

```
proc {RemoveOldNeighbor U W}
 if {Problem2Exist U W} then
   V = {SelectRandom {NewlyAdded U}} in
   {RemoveNeighbor U W}  {RemoveNeighbor U V}
   local Z = {GenerateRandomNeighbor U} in
       if Z == W then {PutNeighbor U W}
       else {PutNeighbor U V} end
   end
end
```

Fig. 4. : Removing old neighbor in parallel rewiring

The time complexity is the same as the previous algorithms. The results of an experimental study of performance in relation to the number of vertexes and the number of machines are depicted in the graph of **Fig. 5.** The performance is measured in seconds. The values shown are the mean values obtained from at least 10 experiments. An experiment is terminated, if it exceeds 300 seconds. The results prove that the developed parallel algorithm gives a reasonable speed up. It also shows that increasing the number of vertexes can be efficiently handled by increasing the number of machines.

Conclusion

The β-model of SW graph could be constructed in parallel while distributed in different machines, using two approaches. The first serializes the rewiring, while the second is fully parallel. The key factor is an efficient parallelization technique to cope with the heavy

messages load. The parallel implementation gives a very good speed up. The parallelization strategy is based on minimizing the overhead of the number of physical messages sent between processes, and second, minimizing the thread creation and management overhead. A detailed version of this paper can be found at www.sics.se/~mahmoud/ICities/papers/ParallelDistributedSmallWorldGraph.pdf.

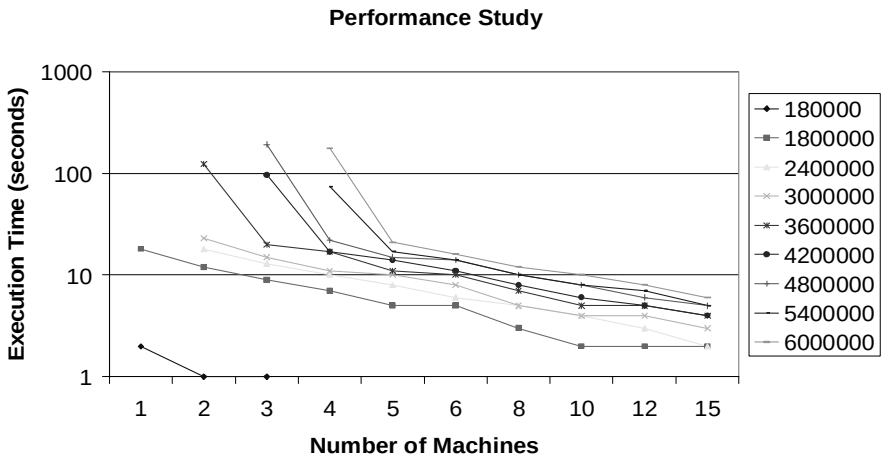

Fig. 5. Performance in relation to the number of vertexes and the number of machines

References

1. Epstein, Joshua M. and Axtell, Robert, 1996, “Growing artificial societies: social science from the bottom up”, The MIT Press, Cambridge, Massachusetts, ISBN 0-262-05053-6.
2. Mahmoud Rafea, Fredrik Holmgren, Konstantin Popov, Seif Haridi, Stelios Lelis, Petros Kavassalis, Jakka Sairamesh, 2002, “Application Architecture Of The Internet Simulation Model: Web Word Of Mouth (WoM)”, IASTED International Conference on Modelling and Simulation (MS2002), May 13-15, 2002, Marina del Rey, USA.
3. Mahmoud Rafea, Konstantin Popov, Fredrik Holmgren, Seif Haridi, Stelios Lelis, Petros Kavassalis, and Jakka Sairamesh, ”large scale agent-based simulation environment”, submitted to Jouranl of Systems Analysis Modelling Simulation.
4. Mozart Consortium, 1999, “The Mozart programming system”, Available at http://www.mozart-oz.org/.
5. Stelios Lelis, Petros Kavassalis, Jakka Sairamesh, Seif Haridi, Fredrik Holmgren, Mahmoud Rafea, & Antonis Hatistamatiou, 2001, " Regularities in the Formation and Evolution of Information Cities", The Second Kyoto Meeting on Digital Cities, October19-20,2001 Kyoto Research Park, Kyoto, JAPAN.
6. Watts, Duncan J., 1999, “Small worlds: the dynamics of networks between order and randomness”, Princeton University Press, New Jersey, ISBN 0-691-00541-9.